AFRL-AFOSR-JP-TR-2017-0015

Verified OS Interface Code Synthesis

Gerwin Klein
NATIONAL ICT AUSTRALIA LIMITED

02/14/2017
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ IOA
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 22-02-2017	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* 30 Sep 2014 to 29 Sep 2016

4. TITLE AND SUBTITLE Verified OS Interface Code Synthesis	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA2386-14-1-4093
	5c. PROGRAM ELEMENT NUMBER 61102F
6. AUTHOR(S) Gerwin Klein, Toby Murray	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NATIONAL ICT AUSTRALIA LIMITED L 5 13 GARDEN ST EVELEIGH, 2015 AU	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AOARD UNIT 45002 APO AP 96338-5002	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/AFOSR IOA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-AFOSR-JP-TR-2017-0015

12. DISTRIBUTION/AVAILABILITY STATEMENT
A DISTRIBUTION UNLIMITED: PB Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The outcome of the project is that they have solved some fundamental challenges, but more work is required to integrate these results into the larger proof framework of the seL4 microkernel to be directly usable in practice. Beyond the stated project goals, the solution approach required
and enabled them to address a more fundamental challenge: the integration of different interactive theorem proving systems without sacrificing correctness guarantees. While again more work needs to be done to fully complete this integration in a way that is usable for a larger audience, this result means that the strong binary verification automation tools from the HOL4 theorem prover, which were used for the verified ML compiler CakeML, can now also be used in the Isabelle/HOL system that was used for the verified seL4 microkernel. This combination
increases proof productivity and enables larger, more expressive systems to be verified in the future.

15. SUBJECT TERMS
Code Verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** SERNA, MARIO
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			
Unclassified	Unclassified	Unclassified	SAR	38	**19b. TELEPHONE NUMBER** *(Include area code)* 315-227-7002

Verified OS Interface Code Synthesis

Final Report for AFOSR AOARD Grant FA2386-14-1-4093

Gerwin Klein, Ramana Kumar, Toby Murray

gerwin.klein@data61.csiro.au

December 2016

Summary

This document forms the final report for the project *Verified OS Interface Code Synthesis*, AFOSR AOARD Grant FA2386-14-1-4093.

The aim of the project was to address the correctness problem for operating system (OS) interface code by automated code/proof co-generation. In particular, we investigated a suitable specification language for OS interface code, automated code generation, and automated proof generation for this generated code.

The outcome of the project is that we have solved the fundamental challenges, but more work is required to integrate these results into the larger proof framework of the seL4 microkernel to be directly usable in practice. Beyond the stated project goals, our solution approach required and enabled us to address a more fundamental challenge: the integration of different interactive theorem proving systems without sacrificing correctness guarantees. While again more work needs to be done to fully complete this integration in a way that is usable for a larger audience, this result means that the strong binary verification automation tools from the HOL4 theorem prover, which were used for the verified ML compiler CakeML, can now also be used in the Isabelle/HOL system that was used for the verified seL4 microkernel. This combination increases proof productivity and enables larger, more expressive systems to be verified in the future.

Contents

1 Introduction

The central question of this project was how to ensure the correctness of Operating System (OS) interface code, using proof-producing code synthesis. Its aim was to demonstrate the technique with a tool for the formally verified microkernel seL4; however its research outcomes are more widely applicable to other OS kernels.

OS interface code sits at the boundary between user-level application programs written in a programming language such as C, and the OS kernel itself, which typically provides a language-independent application binary interface (ABI).

From a system-wide security and safety perspective, this code is part of the low-level trusted computing base of any trusted user-level component in the system. Its correctness is therefore critical to the safe and secure operation of the overall system. Unfortunately, its correctness is also harder than usual to achieve.

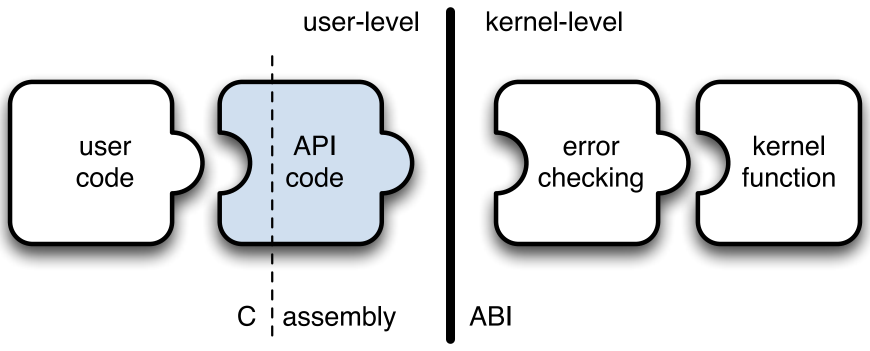

Figure 1.1: OS interface (API) code in context with user and kernel code.

The main reason for this is that such code simultaneously crosses two boundaries: between user-level and kernel-level, and between high-level and assembly-level language. Algorithmically, such interface code is shallow and straight-forward, but the details of compiler calling conventions that bridge, for instance, C to assembly are tricky for humans to adhere to, and kernel ABI specifications can be large and repetitive.

Traditionally, such code is written manually and not formally verified. Although interface code tends to be executed often, and be therefore well tested, its repetitive, shallow, but tricky nature makes human error likely. In our experience the quality of interface code in rarely-used OS functionality is often low and exhibits a higher-than-usual defect rate.

Manual, interactive formal verification, as has been demonstrated for the seL4 microkernel [Klein et al., 2009], is not directly a good fit for this problem for similar reasons: the correctness properties are repetitive and hard to phrase, proofs about low-level assembly are tedious, and since they are highly platform dependent, they will have to be updated often.

The current state of the art, used for instance in the OKL4 and seL4 kernels, is to produce OS interface code automatically from a small domain-specific language (DSL) or XML specification. This reduces the likelihood of human error, but does not eliminate it and makes the code generator part of the trusted computing base. For high-assurance systems, a more robust approach is needed.

This project proposed to not only automatically generate the interface code, but to simultaneously generate the correctness statements as well as corresponding machine-checked proofs of those properties. The result is high-assurance OS interface code with strong formal evidence generated from a high-level specification by fully automatic push-button technology.

This document reports the final outcomes of the project. We begin in Chapter 2 by summarising the original work plan for the project, and in particular the initial approach taken to reason about the combination of C and assembly in which OS interface code is written, using the theorem prover Isabelle [Nipkow et al., 2002, Nipkow and Klein, 2014]. While reporting the results of this work we explain how, while executing that plan, we saw the opportunity to achieve much higher levels of fidelity in reasoning without sacrificing on soundness or trustworthiness, using a different approach. This new approach and its results we explain in Chapter 3. This approach involved carrying out all reasoning at the assembly level after compiling the C portions of the OS interface code, using an external automated proof-producing toolset, in the HOL theorem prover [Slind and Norrish, 2008], then soundly translating those proofs into Isabelle (thereby automatically *reconstructing* them in Isabelle), where they can in future be integrated with theorems proved about the verified microkernel itself.

Chapter 4 summarises the results of the project at a high level, and their potential impact into the future, as well as considering potential future work in this area.

2 Reasoning about C and Assembly Code

In order to motivate the initial approach we took to the automated verification and synthesis of OS interface code, we first discuss the function of OS interface code in Section 2.1. Section 2.3 then describes the domain specific language constructed to synthesise seL4 interface code. In Section 2.4 we discuss the approach we trialled to verify this code. While this approach was very intuitive as we explain in Section 2.5, it presented limitations in terms of compositionality and automation. This ultimately led us to the approach we describe in Chapter 3.

2.1 Background: System Calls

The purpose of OS interface code is to implement the interface between the operating system kernel and application code that runs on top of the kernel. An OS kernel provides a number of services to the applications that it hosts. When an application wishes to request a service from the kernel, it does so by making a *system call*, which causes execution of special instructions that suspend execution of the application. In response, the kernel takes over and begins executing on the application's behalf, in order to provide the service requested by the application. Once the kernel is finished, it returns control back to the application which picks up executing where it left off when it made the system call.

Typical system calls include creating new threads, or configuring access to hardware devices, or communicating with other application components in the system. An important difference between application execution (before and after the system call) and kernel execution (during the system call) is that the former is unprivileged, while the latter is privileged. Indeed, this is why the application must request the service from the kernel by making a system call, because only the kernel is privileged enough to provide it. Forcing the unprivileged application to always go through the kernel also ensures that the kernel can mediate on all interactions between applications and on all device configuration, which is necessary for instance to ensure that mandatory security policies are enforced [Loscocco et al., 1998].

Transferring execution, at the time the system call is made by the application, from the unprivileged application to the privileged kernel necessitates switching the *execution mode* of the processor (in this case we are using the ARMv7 processor architecture). The application accomplishes this task by issuing the `swi` ("software interrupt") instruction, which causes an interrupt to be generated. The CPU receives this software interrupt, and in response switches execution mode to begin executing privileged code located at a fixed address in memory. This privileged code must first determine which particular system call has been requested (e.g. whether `create_thread()` or `send_ipc()` etc.), and *decode* any arguments given to it by the application. To do so, it examines the contents of the CPU registers, as they were left by the application, as well as the contents of specific locations in memory. This means, to pass this information to the kernel, the application places specific values into its registers, and writes certain values into memory at specific locations, before it issues the `swi` instruction.

Once the system call is complete and control returns to the application, it has to do the reverse in order to decode the results of the system call: the kernel leaves these results in specific registers and memory locations, which are interpreted by the unprivileged application code.

At a high level, we say that before the system call the application *marshals* the data describing the system call it is requesting and any arguments to that call to the kernel; after the system call returns the application must *unmarshal* the results of the system call.

2.2 seL4: Microkernel System Calls

On seL4, the verified kernel for which we synthesised the OS interface code for this project, the application/kernel interface through which system call data is marshalled works as follows. Each thread has a dynamically configurable, designed region of memory, called its *inter-process communication (IPC) buffer*.

An application identifies the system call it wishes to perform, and its arguments, to the kernel by placing the data for the call into the application's IPC buffer, and then calling the `swi` instruction. Likewise, the results of the system call form the kernel to the application are placed into the application's IPC buffer by the kernel once the system call has completed.

When marshalling data into the IPC buffer before making the system call, and similarly when unmarshalling the data afterwards, the IPC buffer is interpreted as a set of *message registers* (MRs): a fixed set of word-sized slots in the IPC buffer. Additionally, data for the first few MRs is allowed to be placed in hardware registers for improved performance in the case of system calls with short payloads. In seL4 on ARM, the first four MRs are backed by hardware registers '`r2`', '`r3`', '`r4`' and '`r5`'.

Thus the job of seL4 interface code is as follows.

1. Marshal system call arguments into message registers
2. Issue the `swi` instruction
3. Unmarshal the system call results from the message registers

Steps 1 and 3 are *specific* to each system call, while step 2 is entirely *generic* (i.e. it is the same for every system call). This suggests that we can synthesise the interface code by synthesising that for the specific steps for each system call, and having them call into a fixed subroutine for implementing step 2. This is the approach we took in this project, described in the following section.

2.3 A System Call DSL

While the subroutine for issuing the swi instruction is fixed for every system call, each system call is encoded differently by placing specific values and arguments into the message registers before calling this subroutine. The code for marshalling the system call arguments into the message registers is relatively straightforward in functionality, but tedious and potentially error prone to write manually for every system call. seL4 in particular supports a wide variety of system calls, because it implements a very flexible and fine-grained API in which kernel services are exposed through typed abstractions called *kernel objects*, each of which supports corresponding *methods*. Table 2.1 and Table 2.2 list the 44 seL4 system calls present in the current version of seL4 (reference manual version 4.0.0) on the ARM architecture.

To alleviate this problem, we developed an XML-based domain specific language (DSL) in which each system call could be specified. From this DSL, we then automatically synthesise C code for marshalling and unmarshalling system call arguments and results respectively. Specifically, take the system call TCB - Set Priority from Table 2.1 as an example. Its XML-based DSL description is depicted in Figure 2.1.

```
<api>
  <interface name="seL4_TCB">
    <method id="TCBSetPriority" name="SetPriority">
      <param dir="in" name="priority" type="uint8_t"/>
    </method>
  </interface>
</api>
```

Figure 2.1: The DSL description of the TCB - Set Priority system call.

From this DSL description. we automatically synthesise the definition and implementation of the following C function, seL4_TCB_SetPriority(), whose generated implementation appears in Figure 2.2:

```
int seL4_TCB_SetPriority(seL4_TCB service, uint8_t priority);
```

Lines 3–8 marshal the system call data; line 11 invokes the manually-written seL4_Call() function, which performs the swi; and finally line 13 simultaneously unmarshals the system call results and then returns.

The function seL4_Call() was written manually in a mixture of C and inline assembly. Its compiled assembly implementation is depicted in Figure 2.3. Recall that, in order to allow convenient access by the kernel on the ARM platform, the first four message registers (MRs) are passed between the application and kernel in the physical CPU registers r2, r3, r4 and r5. seL4_Call() performs the same functionality for every system call: it copies the first four MRs into CPU registers r2, r3, r4 and r5 for quick access by the kernel, and then invokes the swi instruction. After the system call has returned, it does the reverse, copying the contents

Object-Type	Method Name
CNode	Cancel Badged Sends
CNode	Copy
CNode	Delete
CNode	Mint
CNode	Move
CNode	Mutate
CNode	Revoke
CNode	Rotate
CNode	Save Caller
Domain Set	Set
IRQ Control	Get
IRQ Handler	Acknowledge
IRQ Handler	Clear
IRQ Handler	Set Notification
TCB	Bind Notification
TCB	Configure Single Stepping
TCB	Configure
TCB	Copy Registers
TCB	Get Breakpoint
TCB	Read Registers
TCB	Resume
TCB	Set Breakpoint
TCB	Set CPU Affinity
TCB	Set IPC Buffer
TCB	Set Maximum Controlled Priority
TCB	Set Priority
TCB	Set Space
TCB	Suspend
TCB	Unbind Notification
TCB	Unset Breakpoint
TCB	Write Registers
Untyped	Retype

Table 2.1: seL4 Architecture Independent System Calls.

of `r2`, `r3`, `r4` and `r5` into the message registers of the thread's IPC buffer and returns the result of the system call.

```
int seL4_TCB_SetPriority(seL4_TCB service, uint8_t priority)
{
    seL4_MessageInfo_t tag =
               seL4_MessageInfo_new(TCBSetPriority, 0, 0, 1);
    seL4_MessageInfo_t output_tag;

    /* Marshal input parameters. */
    seL4_SetMR(0, (priority & 0xff));

    /* Perform the call. */
    output_tag = seL4_Call(service, tag);

    return seL4_MessageInfo_get_label(output_tag);
}
```

Figure 2.2: DSL-Generated implementation of the `seL4_TCB_SetPriority()` function.

```
mvn     r3, #12288              ; 0x3000
ldr     ip, [r3, #-4095]        ; 0xffff f001
push    {r4, r5, r7}
mvn     r7, #0
ldmib   ip, {r2, r3, r4, r5}
swi     0x00ffffff
mvn     r0, #12288              ; 0x3000
ldr     ip, [r0, #-4095]        ; 0xffff f001
mov     r0, r1
stmib   ip, {r2, r3, r4, r5}
pop     {r4, r5, r7}
bx      lr
```

Figure 2.3: The compiled (assembly) implementation of the `seL4_Call()` function.

Object-Type	Method Name
ARM ASID Control	Make Pool
ARM ASID Pool	Assign
ARM Page	Clean Data
ARM Page	Invalidate Data
ARM Page	Clean and Invalidate Data
ARM Page	Unify Instruction Cache
ARM Page	Map
ARM Page	Remap
ARM Page	Unmap
ARM Page	Get Address
ARM Page Table	Map
ARM Page Table	Unap

Table 2.2: seL4 ARM Architecture Dependent System Calls.

2.4 Verification Approach

We synthesise each system call by automatically generating C code (e.g. Figure 2.2) that calls a fixed assembly routine, `seL4_Call()` (see Figure 2.3).

Verifying the system call code therefore requires verifying both the automatically generated C code as well as the implementation of `seL4_Call()`.

Reasoning about `seL4_Call()` has to be performed at the level of its assembly implementation, since it is only at this level that its implementation has meaning.

Therefore, one way to automatically verify the correctness of the synthesised system call code is to:

1. Automatically verify the generated C code for each system call.
2. Verify the fixed assembly implementation of `seL4_Call()` once, manually.

This was the original approach we took for this project. While this approach was ultimately superseded by the work described in the Chapter 3 for reasons we discuss shortly in Section 2.5, we briefly describe each of these steps. In particular, many of the themes encountered in verifying the assembly code for `seL4_Call()`, such as encoding the C compiler's calling convention, reappear in Chapter 3.

2.4.1 C Code Verification

For easy and high-fidelity reasoning about C code, the AutoCorres [Greenaway et al., 2014] tool provides an ideal foundation for reasoning about the generated interface code.

C Code Correctness Code correctness is expressed using *Hoare triples*, which are statements of the form:

$$\{\,P\,\}\,f\,\{\,Q\,\} \tag{2.1}$$

Here, f is a function (e.g. `seL4_TCB_SetPriority()` in Figure 2.2) while P and Q are predicates, called the *precondition* and *postcondition* respectively. Equation 2.1 says that if function f is called in a situation satisfying the condition P then, when f completes, the condition Q will be true. P captures *assumptions* made by the function f, while Q captures the effect of calling f (i.e. f's behaviour).

As an example, consider the trivial function `inc()` implemented in Figure 2.4. This function takes an `int` as its argument and simply increments it, returning the incremented result.

```c
int inc(int x)
{
    return (x + 1);
}
```

Figure 2.4: A trivial C function for incrementing an integer.

The following Hoare triple captures the correctness of this function:

$$\{\,\mathtt{x} < \mathtt{INT\_MAX}\,\}\,\mathtt{inc}(\mathtt{x})\,\{\,\lambda r.\, r = \mathtt{x} + 1\,\} \tag{2.2}$$

Note the initial assumption that the argument `x` be less than the maximum integer `INT_MAX`. Overflow of (signed) integers in C is undefined behaviour, which we must provably rule out if we are to reason formally about C code. This assumption captures the idea that `inc` is correct only when called with an argument which will not overflow when incremented.

The postcondition refers to the function's return value as r and states that it will be one larger than the initial argument x, as expected.

The AutoCorres tool allows one to reason faithfully about C code, including low-level details like the absence of undefined behaviour, integer overflow, memory safety etc. One writes Hoare triples to specify the correctness of their C code, which can then be proved semi-automatically in the Isabelle theorem prover Nipkow et al. [2002]. This provides an extremely trustworthy environment for C code verification.

Reasoning about the Generated C Code Reasoning automatically about the generate C code requires having Hoare triples (correctness statements) proved for the various helper functions that the generated C code calls, such as the fixed function `seL4_SetMR()` called by `seL4_TCB_SetPriority()` in Figure 2.2.

Consider the seL4_SetMR() function as an example for illustration purposes. This function sets the value of a message register in the thread's IPC buffer. Specifically, when

$$\texttt{seL4\_SetMR}(n, v)$$

is called, n denotes the message register that should be set while v is the value that the particular register should be set to. This call sets the value of the nth message register (numbered from 0) to v.

We proved the following correctness statement about the manually-written seL4_SetMR() function, which we can then use to help reason automatically about generated functions like seL4_TCB_SetPriority().

$$\begin{array}{l} \{\, globals\_frame\_intact \;\wedge\; ipc\_buffer\_valid \wedge \\ \quad 0 \leq n \;\wedge\; n < \texttt{seL4\_MaxMsgLength} \;\wedge\; P\;(setMR\;n\;v)\,\} \\ \texttt{seL4\_SetMR(n, v)} \\ \{\, P \,\} \end{array}$$

The first four of the five conjuncts in the precondition are assumptions that seL4_SetMR() makes. Like seL4_Call() (see Figure 2.3), seL4_SetMR() retrieves the pointer to the calling thread's IPC buffer from the *globals frame*, a region of memory at a fixed location in the thread's virtual address space (specifically at address 0xfffff001). The first assumption, *globals_frame_intact* states that this location in memory does indeed hold the address of the thread's IPC buffer. This assumption will always be true in practice since the kernel places the IPC buffer address in this location when it schedules the thread, and ensures that the globals frame is not writable to the thread. However, it must be stated explicitly when reasoning about seL4_SetMR() and the functions that call it, since we reason about these functions in isolation from the kernel in order to keep the reasoning tractable.

The second assumption states that there is a valid IPC buffer at the location pointed to by the IPC buffer pointer: again the kernel will ensure that this requirement is true but it must be stated explicitly in order to reason about seL4_SetMR() in isolation. With these two assumptions, seL4_SetMR() is guaranteed to be able to update the thread's message registers. The third and fourth assumptions then state that the arguments n and v to seL4_SetMR(n,v) are valid, specifically that n is in the range [0,seL4_MaxMsgLength), i.e. identifies a valid message register.

The final conjunct of the precondition is $P\;(setMR\;n\;v)$. It refers to P which is an *arbitrary* postcondition. Specfically, this Hoare triple about seL4_SetMR() is in a form akin to a *weakest precondition* transformer: given some postcondition P that is to be true after seL4_SetMR() is executed, this Hoare triple calculates an appropriate precondition that must be true before seL4_SetMR() is called in order to ensure that the postcondition P is true afterwards.

It states that P must hold for the result of calling the logical auxiliary function *setMR* with the arguments n and v. Here *setMR* calculates the effects of updating the nth message register by setting its value to v: given the current memory contents, it calculates the new memory contents which are identical to the initial one except that the memory location holding the nth message register now holds the value v.

By writing the correctness statement for `seL4_SetMR()` as a weakest precondition rule in this style, we enable automatic reasoning about code that calls `seL4_SetMR()`, such as the generated C code (e.g. the function `seL4_TCB_SetPriority()`).

Reasoning about the generated C code will also requires us to automatically generate Hoare triples that state correctness conditions for functions such as `seL4_TCB_SetPriority()`. We return to this issue later in Section 2.5.

2.4.2 Assembly Code Verification

The `seL4_Call()` function is written in a mixture of C and inline assembly. Unlike for the generated C code, the easiest method of verifying its manually written implementation is at the level of its assembly code semantics (the alternative would be a mixed assembly/C semantics and precise model of the inline assembly code in C, which is possible, but less straightforward). Its principal functions such as copying message registers from the thread's IPC buffer into physical CPU registers, and invoking the `swi` instruction to trigger the system call, which can be given a straightforward semantics at the assembly level, whereas CPU register and instructions like `swi` have no representation at the level of C semantics.

Therefore, to verify this function, we first compile it to assembly as depicted in Figure 2.3. Once represented entirely in assembly code, the semantics of this function become relatively straightforward:

Lines 1–2 Calculate the address of IPC Buffer pointer, place it into the `ip` register (i.e. into `r12`)

Line 3 Save caller's registers (in order to respect the C compiler's calling convention)

Line 4 Place the value -1 (0xffffffff) into register `r7` in accordance with the seL4 system call interface

Line 5 Copy the first four message registers from the IPC buffer to CPU registers `r2`–`r4`,

Line 6 Perform the `swi`,

Lines 7-8 Reload the address of the IPC buffer pointer into the `ip` register

Line 9 Move the system call result value into `r0` so it appears as the return-value of the C function `seL4_Call()` (to respect the C compiler's calling convention)

Line 10 Copy the values from CPU registers `r2`–`r4` into the first four IPC buffer message registers,

Line 11 Restore the caller's registers (in order to respect the C compiler's calling convention)

Line 12 Return to the caller

Of these 12 lines, four of them (namely lines 3, 9, 11 and 12) deal with the interface between C and assembly. The remainder implement the seL4 system call encoding.

The function of each line of assembly that implements the interface expected by the kernel is straightforward to state. Likewise, the conventions that this assembly code must follow to correctly interface to the C functions it calls are also straightforward to state:

- It must save and restore the caller's registers above `r4` that is modifies (namely `r4`–`r7`)
- It must put the return-value into register `r0`
- It must leave registers `r0` and `r1` unmodified before making the `swi` (since these registers hold the values for the C arguments `service` and `tag` to the function `seL4_Call()` — see line 10 of Figure 2.2 — which the kernel expects to appear in these registers)

Verifying this code therefore comes down to being able to give it a sufficiently precise semantics against which these correctness conditions can be stated and then proved.

While the correctness of the generated C code should be automatically stated and proved, we can afford to verify the assembly implementation of `seL4_Call()` manually, since it is itself manually written and fixed for all system calls, i.e. will change only rarely.

To verify this code we first obtained the highly validated and highly detailed model of the ARM assembly instruction set architecture by Fox and Myreen [2010], Fox [2015]. This formal model precisely captures how the ARM CPU decodes and executes each instruction. The model is written in the theorem prover HOL4 [Slind and Norrish, 2008].

To make use of it, we first translated it so that it could be interpreted by the Isabelle theorem prover [Nipkow et al., 2002]. We then stated a Hoare triple expressing the correctness of each individual instruction in the implementation of `seL4_Call()`. We *manually* proved each Hoare triple by reasoning directly over the model of how the ARM ISA interprets each of the instructions. Note that these Hoare triples were stated over the binary encodings of these instructions, and so proving them required first proving that each binary instruction correctly decoded to match the assembly represented in Figure 2.3, and then proving that the effect of each line of assembly was as expected. That is, the final theorem is a Hoare-triple statement about a binary string of bytes.

2.5 Limitations and Discussion

The two-pronged approach, mixing C and assembly verification, provided a useful proof-of-concept that OS interface code could indeed by synthesised and its correctness stated and proved with some degree of automation.

However, we ultimately abandoned this approach in favour of the work described in Chapter 3 due to the non-trivial limitations that we now describe.

Generating Hoare Triple Correctness Statements One of the benefits of this approach is that it allows reuse of existing, automatic C verification tools to automatically verify the generated C code. However, this comes at the cost of having to state the correctness of the generated code at this level of abstraction. Specifically, one must ultimately generate Hoare triples for the entirety of the generated C functions, e.g. `seL4_TCB_SetPriority()`. Such a Hoare triple of the form:

$$\{\, P \,\}\ \texttt{seL4\_TCB\_SetPriority}(\texttt{service}, \texttt{priority})\ \{\, Q \,\}$$

captures assumptions that this function makes in the precondition P. However, and as it turned out problematically, its postcondition Q must capture not only the correct marshalling/unmarshalling of the system call data by the generated C code, but also the effects of the system call itself. The system call semantics is described by the seL4 specification, and is itself a large and complex formal artefact.

When working at the level of the C semantics, it is very difficult to disentangle these two. This means each individual correctness specification for the generated C code would have to make statements about the seL4 kernel behaviour and about interface code at the same time, which makes it very difficult to generate these automatically.

Composing C and Assembly Proofs Another limitation of this approach is that the proofs about the C code and those about the assembly implementation of `seL4_Call()` remain separate from each other, even though all proofs are carried out in the Isabelle theorem prover. The first proofs talk about C semantics, while the second talk about the ARM ISA semantics.

Naturally the latter must talk about details of the C compiler's calling convention, as discussed above; however, those same details are not reflected in the C code semantics itself since it purposefully abstracts away from those details.

Thus this approach leaves two sets of proofs: automatically generated ones about generated C code, and manual ones about manually-written code, neither of which is formally connected to the other. This inability to compose the proofs about the assembly code with those about the generated C code reduces the trustworthiness of this approach, a theme we return to later in this section.

Difficulty of Manual Reasoning over Assembly Model A third difficulty is the high level of effort required to reason manually about the ARM implementation of `seL4_Call()`. Once the ARM ISA model is translated from its original representation in the HOL4 prover, into the Isabelle prover, we no longer have access to an existing and powerful set of tools for carrying out proofs automatically over this model.

While this reduction in efficiency is not fatal, it does increase the difficulty of composing the Hoare triples about each individual instruction: a task that is already automated in the original model in the HOL4 theorem prover.

Trustworthiness of the Assembly Model One final limitation of this approach is the translation of the ARM ISA model from HOL4 to Isabelle. Unlike the approach we describe in the following chapter, this original translation does not certify that the ARM model when translated into Isabelle has the same meaning as it did originally in HOL4. This is unfortunate since the model in HOL4 has been extensively validated by Fox and Myreen [2010], and so is rightly considered extremely trustworthy.

Even though the translation to Isabelle is automatic, there is still the chance of it introducing errors in the model in Isabelle. Since the value of the proofs about the ARM implementation of `seL4_Call()` ultimately rest of the strength of the ARM ISA model, increasing the level of trust in that model is highly desirable.

Summary Ultimately, trying to give OS interface code a semantics at the C level is suboptimal. This realisation led us to pioneer a different approach, which we describe in the following chapter. This approach is based on the realisation that OS interface code can be given a high-fidelity semantics entirely at the level of its assembly code, and that we can extend and apply state-of-the-art methods for importing proofs from one theorem prover to another to allow us to reason automatically at the assembly level in HOL4 and then reuse those results in Isabelle without reducing trust.

3 Reasoning Automatically About Assembly

As described in Chapter 2, the functionality of OS interface code is conceptually and algorithmically very simple. Essentially, it translates system calls made in the user-level program's calling convention to the low-level system call interface provided by the OS kernel. This involves marshalling input parameters into the expected registers, making an appropriate invocation of the underlying system-call interface, and un-marshalling any results. Due to its high-level simplicity, it is unsatisfying to axiomatise a high-level specification of the interface code functionality: potential bugs are invisible at all but the lowest levels of abstraction.

To formally model this functionality with sufficient fidelity to catch errors, our new approach is to work directly at the assembly level, and ensure the assembly code being modelled is generated directly from the code that actually runs. By reasoning entirely at the assembly level, we address the first two limitations described in Section 2.5, namely, the difficulty in generating Hoare triples for C code that must capture the effects of assembly-only functionality (marshalling/unmarshalling), and the difficulty in composing C and assembly proofs. Fortunately, we can leverage existing tools for automatic reasoning about assembly code, in particular, the machine-code decompilation toolchain by Magnus Myreen [Myreen et al., 2012, Myreen, 2008] which is integrated with high-fidelity machine-code ISA models by Anthony Fox [Fox, 2003, Fox and Myreen, 2010].

The machine-code decompilation toolchain is implemented in the HOL theorem prover [Slind and Norrish, 2008], also known as HOL4, which provides a logic and reasoning framework compatible to the Isabelle/HOL theorem prover in which the bulk of the seL4 verification was done. By using the toolchain in HOL4, we also overcome the last two limitations described in Section 2.5, namely, the difficulty of manual reasoning over assembly code, and the trustworthiness of the machine model. This is because the HOL4 toolchain is automatic and extensively validated.

Using the HOL4-based toolchain for reasoning about assembly code, we reduce the problem of reasoning about the seL4 OS interface code to the following tasks:

- Produce assembly code for the OS interface code (§3.1). This assembly code needs to be in a form suitable for machine-code decompilation.
- Run the decompiler to produce low-level theorems about the functionality of the OS interface assembly code (§3.2).
- Transfer the proofs produced in the previous step from HOL4 to Isabelle/HOL (§3.3). This involves proof-recording on the HOL4 side, and proof-replay on the Isabelle/HOL side, as well as theory alignment work to bridge the gaps and differences in theory libraries between the two theorem-proving systems.

- Using the low-level assembly-code functionality theorems as a basis, prove a higher-level statement about how making a call to the OS interface interacts with the assumptions of the OS kernel proof (future work).

The third step is, perhaps surprisingly, the most involved one. However, much of the difficulty is in building proof-transfer technology which is general-purpose and, since it solves the more fundamental problem of sound proof transfer between different theorem proving systems, can be re-used much more broadly in other projects.

3.1 Compiling the Interface Code to Assembly

As mentioned in Section 2.4, the seL4 interface code is a mixture of C code (synthesised from the XML-based DSL) and assembly code (manually written). We would like to reason about this code uniformly at the assembly level, which is the highest level where all of the code has direct meaning. Our approach is to use the C compiler, which takes the mixed-level inputs and produces a uniform binary, and then decompile the result into logic. The advantage of using the compiler is that it is entirely automatic, so scales to handle changes in the synthesised C code for different platforms, and it is fully realistic. The snag is that OS interface code is typically marked as `inline`: in the final binary, the functionality of the interface code is spread to its call sites and not easily separable into an independently verifiable function.

For the purpose of decompilation and verification, we can turn this inlining off and force the compiler to produce an independent function for each item in the OS interface. This makes the verification approach via decompilation possible. It is also possible to run systems with this inlining turned off, although we would expect a performance hit. From the correctness perspective, the difference between reasoning about inlined code and non-inlined code is easy to delineate: if we trust the compiler's inlining facility, then verification of the non-inlined code carries over to systems with the inlining turned on. Compared to the old approach, this is not a reduction in trustworthiness: the old approach already trusted the compiler to be correct — the current approach reduces this need for trust, or enables us to remove it completely for a performance trade-off (turning off inlining).

Figure 3.1 shows an example of the result of compilation with inlining turned off, for the function `seL4_TCB_SetPriority()`. Although we do not inline `seL4_TCB_SetPriority()` itself at its calls sites, we do still inline calls within its definition. The bulk of the code is made up of the (now inlined) call to `seL4_Call()`. However, importantly, the marshalling code (essentially, an inlined call to `seL4_SetMR()`) is also included. Note that some of the assembly mnemonics here are newer synonyms compared to Figure 2.3, e.g., `svc` instead of `swi`.

```
674:  e92d40b0   push    {r4, r5, r7, lr}
678:  e3e0ea03   mvn     lr, #12288 ; 0x3000
67c:  e1a02001   mov     r2, r1
680:  e51ecfff   ldr     ip, [lr, #-4095] ; 0xfffff001
684:  e3e07000   mvn     r7, #0
688:  e28c3008   add     r3, ip, #8
68c:  e58c1004   str     r1, [ip, #4]
690:  e59f1014   ldr     r1, [pc, #20] ; 6ac <seL4_TCB_SetPriority+0x38>
694:  e8930038   ldm     r3, {r3, r4, r5}
698:  efffffff   svc     0x00ffffff
69c:  e51ecfff   ldr     ip, [lr, #-4095] ; 0xfffff001
6a0:  e1a00621   lsr     r0, r1, #12
6a4:  e98c003c   stmib   ip, {r2, r3, r4, r5}
6a8:  e8bd80b0   pop     {r4, r5, r7, pc}
6ac:  00006001   .word   0x00006001
```

Figure 3.1: The compiled (assembly) implementation of `seL4_TCB_SetPriority()` with inlining off.

3.2 Decompilation into Logic

The machine-code decompilation framework [Myreen et al., 2012] automates the process of composing Hoare triples for individual assembly instructions, to obtain a single Hoare triple for a sequence of assembly code. This includes decoding the binary instructions to their assembly representation. Thus, the automatic framework in HOL4 replaces the manual work for verifying assembly code described in Section 2.4.2. The Hoare triple for each instruction is generated from the high-fidelity and well-validated ARMv7 ISA model in HOL4, and the composition is performed by (automatic) forward reasoning within the theorem prover, yielding trustworthy results.

The decompilation framework does not support the `swi` (a.k.a. `svc`) instruction that signals an interrupt to switch the machine into privileged mode. Therefore, for each OS interface function, we produced two composed correctness theorems: one for the snippet of assembly code before entering the kernel via `svc`, and one for the snippet of assembly code after the kernel returns. This is an appropriate split for the interface code correctness theorems, since execution of the seL4 kernel is verified separately at a higher level of abstraction. This split also neatly delineates a limitation of the previous approach: it is precisely the interface point between automatically generated specification for interface code and the high-level manual specification of seL4 behaviour.

An example of the kind of theorem generated by the framework, in particular as produced for the first half of `seL4_TCB_SetPriority()` (before the `svc` instruction) is shown in Figure 3.2 below. The top-level assertion in this theorem is of the form `TRIPLE` $M\ \ state\ \ code\ \ state'$, which essentially asserts that $code$ transforms $state$ to $state'$, according to the machine model M. The code is represented as a set of bytes and their locations in memory, so for example the numbers `(1652w,0xE92D40B0w)` indicates byte `e92d40b0` at location 1652; note that the code is exactly as listed in Figure 3.1. The variables appearing in $state$ consist of a Boolean assertion, pre, which captures the weakest precondition for the Hoare triple to hold, and the state of the machine: the program counter p, the other registers, the memory, and the subset of memory considered to be the stack. The expression denoting $state'$ builds up the required precondition step by step, for example, the first `let` requires the initial program counter p to be set to 1652. There is approximately one `let` for each instruction in straight-line assembly code, each of which captures local changes to the machine state for that instruction as well as collecting the required preconditions.

These theorems capture a high-fidelity but low-level semantics for straight-line assembly code. Importantly, they are generated automatically and are assertions over a validated machine model (represented by `(arm_assert,ARM_MODEL)` in Figure 3.2). It would be easy to post-process these theorems in a theorem prover (either HOL4, where they are generated, or Isabelle/HOL) to, for example, collect the preconditions for individual instructions spread over several `lets` into a single precondition for the whole block; we have not yet attempted any such post-processing, which would be the first step of integrating the generated specifications into the existing seL4 proofs. By manipulating such theorems directly within a theorem prover, we maintain a strong semantic link back to the validated machine model.

3.3 Transferring Results Between Theorem Provers

Once we have produced the `TRIPLE` theorems described in the previous section for each of the seL4 interface functions (Table 2.1 etc.), we ultimately hope to post-process them and connect them to the seL4 model in Isabelle/HOL. The theorems are proved by custom automation (the decompilation toolchain) in HOL4, and HOL4 implements a closely related logic to that of Isabelle/HOL, namely, classical higher-order logic. In this section, we describe our methodology for automatically porting proofs from HOL4 to Isabelle/HOL based on recording and replaying proof traces (Section 3.3.1) and producing portable packages in OpenTheory format [Hurd, 2011] (Section 3.3.2). As part of this work, we developed infrastructure for importing OpenTheory packages into Isabelle/HOL. We also made progress (Section 3.3.3) on making the results of such imports idiomatic in the target prover (Isabelle/HOL), which is a necessary step to facilitate connecting imported theorems with existing developments.

```
⊢ TRIPLE (arm_assert,ARM_MODEL)
     (pre,p,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,n,z,c,
      v,mode,dmem,memory,dom_stack,stack)
     {(1652w,0xE92D40B0w); (1656w,0xE3E0EA03w); (1660w,0xE1A02001w);
      (1664w,0xE51ECFFFw); (1668w,0xE3E07000w); (1672w,0xE28C3008w);
      (1676w,0xE58C1004w); (1680w,0xE59F1014w); (1684w,0xE8930038w);
      (1692w,0xE51ECFFFw); (1696w,0xE1A00621w); ...}
     (let (pre,p,r13,stack) =
            (pre ∧ (p = 1652w) ∧ aligned 2 r13 ∧
             r13 - 16w ∈ dom_stack ∧ r13 - 15w ∈ dom_stack ∧
             r13 - 12w ∈ dom_stack ∧ r13 - 8w ∈ dom_stack ∧ ...
             ∧ r13 - 1w ∈ dom_stack,1656w,
             r13 - 16w,
             WRITE32 (r13 - 16w) r4
               (WRITE32 (r13 - 12w) r5
                  (WRITE32 (r13 - 8w) r7
                     (WRITE32 (r13 - 4w) r14 stack)))) in
      let (pre,p,r14) = (pre ∧ (p = 1656w),1660w,0xFFFFCFFFw) in
      let (pre,p,r2) = (pre ∧ (p = 1660w),1664w,r1) in
      let (pre,p,r12) =
            (pre ∧ (p = 1664w) ∧ aligned 2 (r14 - 3w) ∧
             r14 - 4095w ∈ dmem ∧ r14 - 4094w ∈ dmem ∧
             r14 - 4093w ∈ dmem ∧ r14 - 4092w ∈ dmem,1668w,
             READ32 (r14 - 4095w) memory) in
      let (pre,p,r7) = (pre ∧ (p = 1668w),1672w,0xFFFFFFFFw) in
      let (pre,p,r3) = (pre ∧ (p = 1672w),1676w,r12 + 8w) in
      let (pre,p,mem') =
            (pre ∧ (p = 1676w) ∧ aligned 2 r12 ∧ r12 + 4w ∈ dmem ∧
             r12 + 5w ∈ dmem ∧ r12 + 6w ∈ dmem ∧ r12 + 7w ∈ dmem,
             1680w,WRITE32 (r12 + 4w) r1 memory) in
      let (pre,p,r1) = (pre ∧ (p = 1680w),1684w,24577w) in
      let (pre,p,r3,r4,r5) =
            (pre ∧ (p = 1684w) ∧ r3 ∈ dmem ∧ aligned 2 r3 ∧
             r3 + 1w ∈ dmem ∧ r3 + 2w ∈ dmem ∧ ...
             ∧ r3 + 11w ∈ dmem,1688w,READ32 r3 mem',
             READ32 (r3 + 4w) mem',READ32 (r3 + 8w) mem') in
        (pre,p,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,n,z,
         c,v,mode,dmem,mem',dom_stack,stack))
```

Figure 3.2: Generated correctness theorem for the first half of `seL4_TCB_SetPriority()`.

3.3.1 Proof traces

Both of the theorem provers we are concerned with, HOL4 and Isabelle/HOL, follow the *LCF-style* architecture. This means that every theorem produced by the system ultimately originates from calls to a small set of primitive inference rules implemented in the *logic kernel* of the theorem prover. The primary purpose of the LCF architecture is trustworthiness: one only needs to ensure that the relatively small logic kernel is implemented correctly to obtain, by an architectural argument, soundness of the whole theorem proving system.

However, another ramification of the LCF architecture is that every theorem produced by the system has a proof represented by the sequence of calls to inference rules in the logic kernel that led to that theorem's creation. By instrumenting the kernel to record these calls, we obtain a *proof trace*. Provided the inference rules of one theorem prover's logic kernel can be simulated in another prover, these proof traces can be replayed in the second prover to produce a copy of the theorem in the second prover without compromising the soundness guarantees of the LCF architecture. Using proof traces to record and replay proofs is a well-known approach for LCF-style provers; see, for example Kaliszyk and Krauss [2013].

The novelty in our investigation, therefore, is not in using proof traces for theorem transfer, but in identifying and beginning to address the fact that proof traces are not enough. As described in the following sections, the problem is in ensuring imported theorems connect usefully to natively proved theorems in the target theorem prover. Proof traces do not, on their own, handle the variation in theory libraries and mathematical concepts between different theorem provers.

3.3.2 OpenTheory portable theory packages

The OpenTheory project[1] aims to support sharing formal developments between all theorem provers based on (classical) higher-order logic. The project consists of three aspects. The first is a prover-independent proof-trace format for higher-order logic proofs. The second is the concept of a *theory package*: each package is a formal derivation of results (new theorems and definitions) from assumptions (previously-proved theorems and previously-defined constants). Theory packages can be composed: the results of one package can be mapped onto the assumptions of another, via a *theory interpretation* that renames constants appropriately, to produce a composed package with the intermediate assumptions removed. Thus a package is either atomic (corresponding directly to a recorded proof trace) or a composition of multiple packages. The final aspect of OpenTheory is a standard, well-organised namespace for commonly defined constants and a standard library of commonly used theories.

It is these latter two aspects, compositionality and a standard library, that make OpenTheory an

[1]http://www.gilith.com/research/opentheory/

appealing basis for long-term proof-sharing infrastructure. The typical approach to proof transfer in the past has involved engineering a one-off proof-trace based solution fit to the task at hand. Inevitably, such solutions become deprecated as the formal developments they are connecting undergo further development. By aiming for portable theory packages and re-usable infrastructure based on a common format, we think that a connection based on OpenTheory will be more reliable and sustainable.

The machine-code decompilation toolchain in HOL4 depends on a large fraction of the standard libraries included in the HOL4 distribution, including, for example, theories of integers, floating-point numbers, fixed-width words, as well as basic theories for Booleans, natural numbers, and lists. It also, of course, depends on the substantial formalisation of the ARM machine ISA model. Therefore, as a pre-requisite to recording re-usable proof traces for the assembly-code theorems from Section 3.2, we first created OpenTheory packages for all its dependencies in the HOL4 distribution. In the remainder of this section, we describe at a high level how we created and organised these packages, and how we import them into Isabelle/HOL.

Exporting HOL4 library packages To export OpenTheory packages from the HOL4 prover, we use a switch in the HOL4 logic kernel to turn on proof-trace recording. (We built on and improved prior work by the second author on recording HOL4 proofs in the OpenTheory proof-trace format.) Initially, we record proofs exactly as they occur in HOL4, with HOL4-specific constant names. This can be done automatically, without touching the proof script files used for building the standard HOL4 distribution. Subsequently, we use theory interpretations to map constants, wherever possible, into the OpenTheory standard namespace. To achieve this post-recording processing, we use the existing infrastructure provided by the OpenTheory project in the `opentheory` tool.

Some of the logical content of the HOL4 libraries is already covered by the OpenTheory standard library. The standard library, whose package name is `base`, aims to be the intersection of libraries supported by all the major HOL-based provers. In support of our aim to produce reusable theory packages, we would rather not produce HOL4-specific derivations of this logical content, which would deviate from the standard library's organisation. However, this requires a little more care than simply recording the HOL4 libraries as they are.

Our solution is to create a bridging theory, called `hol-base`, that assumes the results provided by `base` and proves the results required by the rest of the HOL4 dependencies. The construction of `hol-base` involves importing the results of `base` into HOL4, calculating the set of theorems used by the HOL4 library that are not already provided by `base`, and re-proving those theorems using only `base` theorems as assumptions. The final step in this construction is currently a little tedious, however we believe there is room for further automation. The approach has good leverage in that the HOL4 library packages (which now depend only on `base` via `hol-base`) become reusable by any theorem prover supporting `base`.

The packages resulting from this work are now available online as part of a public OpenTheory repository[2]. In particular, see the `machine-code-straightline` package and its dependencies (which include the ARM model).

Importing OpenTheory packages into Isabelle/HOL To import OpenTheory packages, one starts with its proof trace. (The `opentheory` tool can produce a proof trace for a composed package by stitching together the traces for the underlying atomic packages.) The semantics of an OpenTheory proof trace[3] is specified by an abstract machine representing a generic theorem prover's logic kernel. We simulate each of the inference rules of this virtual machine using Isabelle/HOL's logic kernel, which is reasonably straightforward.

OpenTheory is designed to be easy to import for any theorem prover with a native version of the logical content of the `base` standard library package. A little care is required to target the correct native constants in Isabelle/HOL when replaying OpenTheory proofs. For example, the `base` package includes a constant called `List.reverse` and various theorems about it, which can be referred to by packages that depend on `base`. When replaying such a package, the import machinery needs to replace any reference to `List.reverse` with the correct function (`rev`) in the native Isabelle/HOL library. When a theorem about `List.reverse` is required during replay, it may need to be proved either on-the-fly automatically (from existing theorems in Isabelle/HOL's database) or handed back to the user for a (usually simple) manual proof. This process is essentially the inverse of constructing the bridging theory (`hol-base`) for export.

For simple renamings (such as `List.reverse` to `rev`), replacement during import is sufficient. However, in some cases there are more significant deviations between the formal developments of some concept in different theorem provers' libraries. We turn to some examples of these alignment problems, and our ideas for handling them, in the next section. Up to alignment, we have successfully imported all the automatically produced assembly-code theorems (such as in Figure 3.2) from HOL4 into Isabelle/HOL, via proof-trace replay of reusable OpenTheory packages.

3.3.3 Aligning theory libraries

To illustrate the alignment problem, let us consider first a simple example: natural number numerals. The standard definition of natural numbers in higher-order logic is, essentially, an inductive datatype with constructors `0` and `Suc`. Considered as numerals, this is a unary representation: the size of a term representing a number grows in linear proportion to the size of the number, e.g., 4 is represented as `Suc(Suc(Suc(Suc(0))))`. This is inefficient—large term sizes slow down interactive theorem proving—so most theorem provers also provide a

[2]http://opentheory.gilith.com/packages/

[3]documented at http://www.gilith.com/research/opentheory/article.html

binary numeral representation and normalise terms to use it wherever possible. The precise choice of representation, however, is a design decision with multiple options. Isabelle/HOL uses traditional bits (`bit0` and `bit1`), where 4 is `bit0(bit0(bit1(0)))`, whereas HOL4 uses an alternative encoding (`bit1` and `bit2`) where 4 is `bit2(bit1(0))`. The OpenTheory standard library uses the traditional representation like Isabelle/HOL.

When exporting HOL4 theories as OpenTheory packages on top of `base`, we must choose whether those packages should define the `bit2` constant and represent numerals using it, or whether work should be done to convert all numerals into the traditional format. If we create packages that expose `bit2`, then on import into Isabelle/HOL the resulting theorems will not be compatible with much of Isabelle/HOL's infrastructure (e.g., the simplifier, the term pretty-printer, and numerical decision procedures) that expects numerals in the traditional format. However, expunging `bit2` entirely from proofs on the HOL4 side might require significant engineering.

Our solution in this case is to leave most of the HOL4 proofs intact, but do some additional automatic forward proof on just the top-level desired theorems (the `TRIPLE` theorems for the OS interface assembly functions). The job of the automation is to rewrite numerals from one format into the other. More specifically, we define a version of `bit0` in HOL4 and write a small piece of automation to prove an equation between a HOL4 numeral and its traditional counterpart, which can then be used as a rewrite rule. The result is that the OpenTheory packages, and the theorems imported into Isabelle/HOL, do refer to `bit2`, but only inside proofs: the theorems of interest have `bit2` removed and are suitable for further processing by native Isabelle/HOL tools.

Our solution for numerals is possible because the change in representation happens within a single type (natural numbers), which is shared between both provers (and the OpenTheory standard library). Specifically, this allows us to make local changes to theorems at the relevant subterms. Things become more difficult when the difference between the two libraries is more substantial as in the following.

Our second example is the representation of sets: in HOL4, sets are identified with predicates, so a set with elements of type `'a` is represented by its characteristic function of type `'a => bool`; in Isabelle/HOL, sets are represented directly by a new type, i.e., `'a set`. In this case also, the OpenTheory standard library matches the Isabelle/HOL convention. Recall that in an assertion of the form `TRIPLE` $M\ state\ code\ state'$, the $code$ is represented as a set of (address, encoded instruction) pairs. Thus in HOL4, the third argument to `TRIPLE` is a predicate, namely, the characteristic function of the $code$ set. On import into Isabelle/HOL, we expect constants like `TRIPLE`, which are specific to the machine-code decompilation toolchain, to be defined as part of the proof replay. However, to be able to use Isabelle/HOL's native set operators (union, intersection, etc.), we want the third argument to `TRIPLE` to be a set, not a predicate.

We have not yet implemented any approach to resolving alignment problems of this nature, and would hope to investigate them as future work. Ideally, we would like to treat sets in a similar way to numerals: with a little bit of additional forward proof prior to export into OpenTheory. This is desirable because it is unintrusive—the original proofs stay intact—and separates concerns. However, we cannot get away with a simple rewrite rule: in this case, an alternative version of the `TRIPLE` constant needs to be defined at some point, with a different type from the original. We expect finding a neat solution to this kind of alignment problem to substantially improve the viability of reliable proof transfer between different theorem proving systems.

4 Conclusion

4.1 Summary of Results

The project set out to automatically produce OS interface code, to automatically produce OS interface code specifications, and to automatically produce proofs that the code satisfies these specifications. The project additionally had the stretch goal of integrating these proofs with the seL4 microkernel specification.

We achieved the three main goals first in the originally proposed mixed C/assembly approach, and, after recognising a number of limitation of this approach, a second time in a more trustworthy, flexible and foundational pure assembly verification approach, using sound theorem transfer from the HOL4 to the Isabelle/HOL proof system. We did not yet achieve the stretch goal of proof integration with seL4, but we believe that the new approach provides the necessary foundations for doing so.

4.2 Implications of these Results

The current state of the work means that we can automatically get proved properties about microkernel OS interface code in the theorem prover Isabelle/HOL based on a highly validated and trustworthy assembly verification engine in the HOL4 theorem prover.

This in turn means that we can now reason about the effects of such interface code formally in Isabelle/HOL.

With code, specification, and proofs all being generated from the same set of tools, there is a remaining risk that all three artefacts reflect exactly the same unexpected behaviour. However, the proof chain means that any deviations from expected behaviour are necessarily reflected in the specification. That is, defects cannot remain hidden in inscrutable generated code, but are exposed at the specification levels, and validation can be reduced to this specification level.

Integration with the rest of the seL4 proofs in future work would provide such additional validation for the generated specifications and give additional assurance that they correctly reflect the expected behaviour.

Beyond the immediate scope of this project of OS interface code, our new solution approach has more wide-ranging implications for the power of automated reasoning tools available for interactive software verification. While sound theorem transfer between the HOL4 and Isabelle/HOL systems has been achieved in the past for smaller artefacts, this is the first time we are aware of that an industrial-sized specification such as the Cambridge ARM model and powerful reasoning tools such as the ones by Myreen et al. [2012] have been used successfully for a real verification. Our existing collaborations with Cambridge and the OpenTheory team were instrumental in achieving this connection.

We think this result provides exciting opportunities for further work in increasing the reach of reasoning tools for high-assurance software development.

4.3 Future Work

While the project was more successful than initially planned for in terms of reaching its main objectives twice and pioneering a strong usable connection between two of the leading theorem proving systems in the field, there is more work required to fully complete the objective of fully automated high-assurance interface code.

The main two areas for further work are:

- **Proof Integration with seL4.** As indicated above, while the main goals of the project were achieved and we demonstrated that the approach works, the stretch goal of integrating the OS interface proofs with the seL4 proof was not. Doing so would be necessary to use the results in practice for connecting new user-level proofs to the existing kernel-level proofs to achieve end-to-end high-assurance systems. It would also provide the necessary validation that the generated specification artefacts do reflect the expected behaviour of the interface code.
- **Complete Tool and Library Integration between HOL4 and Isabelle/HOL.** The exciting new proof-preserving connection via OpenTheory from HOL4 into Isabelle/HOL that this project pioneered is at a stage where we can show that the approach works for this particular case, but more work is needed to demonstrate that this kind of connection can achieve a full fusion of the reasoning power of the two proof systems. Previous approaches demonstrated proof transport between the systems in principle, but could not overcome the problem of library mismatch and concept mismatch between the systems. Our approach has the concrete potential for doing that and for achieving a lasting impact on the ability of the field to increase the productivity of formal verification engineers in high-assurance software.

List of Figures

Bibliography

Anthony Fox. Formal specification and verification of ARM6. In *16th TPHOLs*, volume 2758 of *LNCS*, pages 25–40, Rome, Italy, Sep 2003. 21

Anthony Fox. Improved tool support for machine-code decompilation in HOL4. In *ITP*, volume 9236 of *LNCS*, pages 187–202, 2015. 18

Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st ITP*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, Jul 2010. 18, 20, 21

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *PLDI*, pages 429–439, Edinburgh, UK, Jun 2014. 14

Joe Hurd. The OpenTheory standard theory library. In *NFM*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191, Apr 2011. 24

Cezary Kaliszyk and Alexander Krauss. Scalable LCF-style proof translation. In *ITP*, volume 7998 of *LNCS*, pages 51–66, 2013. 26

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. 7

Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, Jeff Turner, and John Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Inform. Syst. Security Conf.*, pages 303–314, Oct 1998. 9

Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, Computer Laboratory, Cambridge, UK, Dec 2008. 21

Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic – improved. In *2012 FMCAD*, pages 78–81, Cambridge, UK, Oct 2012. 21, 23, 32

Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. 2014. 8

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. 2002. 8, 15, 18

Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, pages 28–32, Montréal, Canada, Aug 2008. 8, 18, 21

CONTACT US

t 1300 363 400
+61 3 9345 2176

e enquiries@data61.csiro.au

w www.data61.csiro.au

AT CSIRO WE SHAPE THE FUTURE

We do this by using science and technology to solve real issues. Our research makes a difference to industry, people and the planet.

FOR FURTHER INFORMATION

Gerwin Klein, Ramana Kumar, Toby Murray

t +61 2 8306 0550

e gerwin.klein@data61.csiro.au

w trustworthy.systems